IOFSL Design Document

Kamil Iskra, Rob Ross, Dries Kimpe, Rob Latham, Sam Lang, Phil Carns, Nawab Ali

January 12, 2009

1 Introduction

This document is intended as an extension of the proposal document [8] for the I/O Forwarding Scalable Layer (IOFSL) project [3]. It outlines the design decisions made until now in the implementation process. We provide the rationale for these decisions and discuss their consequences.

A high-level overview of the software stack can be found in Figure 1.

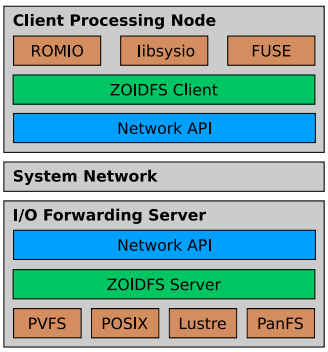

Figure 1: Software organization across client processing nodes and I/O forwarding servers.

2 Networking API

We have opted for BMI [1] as the networking API between the compute nodes and the I/O forwarding nodes. The following characteristics of BMI contributed to this decision:

- light-weight,
- asynchronous API,
- portable,
- field-tested in HEC environment,
- in-house expertise available.

BMI is used in PVFS; it is in fact distributed as part of PVFS. We made changes to ease the extraction of BMI from PVFS; these changes have been integrated into PVFS 2.8.0. A BMI distribution can be created by configuring PVFS with `--enable-bmi-only`, and built with `make dist` to obtain a standalone copy of the BMI library.

As indicated above, BMI is highly portable: it supports TCP/IP sockets, Infiniband, MX, and Portals. That covers most of our target platforms of interest, with one major exception: the collective network of IBM Blue Gene. Blue Gene's

network interfaces are proprietary and employ custom network protocols. However, we have an in-house expertise in programming the collective network, and will in due time implement the necessary support for the Blue Gene in BMI. For the time being, on the Blue Gene we use an alternative, platform-specific I/O forwarding infrastructure, called ZOID [6]—part of the ZeptoOS project [9].

Using BMI for I/O forwarding can lead to some complications if we also use PVFS on the I/O forwarding nodes. Namely, we could end up with two copies of the BMI library within the I/O forwarding daemon process: one for I/O forwarding, the other for PVFS. This could potentially lead to clashes. For now, we require that the same copy of BMI library be used for both purposes. We could consider breaking BMI out into a separate library in PVFS to make this easier. If that turns out to be impractical, we can employ compiler and linker tricks to limit the visibility of symbols from one of the libraries so that two copies of BMI can function within one process.

3 I/O Forwarding Protocol

We defined a new I/O forwarding protocol, called ZOIDFS, which is more suitable for the task than the common POSIX file I/O. It is a stateless protocol, reminiscent of NFSv3. Instead of file descriptors, it has opaque, 16-byte file handles, which can be freely exchanged between compute processes, as there is no state associated with them. The API is more flexible and more expressive; fewer calls are needed to achieve the same goal. For example, the lookup call (an equivalent of open) accepts either a full pathname or a parent handle and a component name, while the data read/write calls can operate on multiple memory buffers and multiple regions of the file within a single call.

The API has so far remained remarkably stable; we have not added any new calls or even any new parameters to existing calls. Having said that, our implementation efforts are not yet complete, so the API could yet change.

We have identified a need for one extension: an ability to pass *hints* along with the API function calls, which would provide contextual information helpful for optimizations or debugging. Candidates to be passed this way include:

- node id,
- process id,
- operation id,
- user credentials.

The idea behind the *operation id* is to be able to identify individual sub-operations coming from multiple compute processes, that form a larger, application-wide collective operation. This could be helpful to a separate caching layer running on the I/O forwarding nodes. Note that building such a caching layer is a research project, and is not part of the IOFSL project.

This feature is still in early design stages; work on implementing it has not started yet, and the above list is preliminary.

4 Call Forwarding

Call forwarding is the central component of this project, and consists both of the code running on the compute nodes that marshalls arguments of ZOIDFS calls and demarshalls the results, and of a corresponding daemon running on the I/O forwarding node that replays those calls.

We have opted to implement call forwarding from scratch, rather than extend the existing infrastructure provided by ZOID. The following factors contributed to this decision:

- ZOID client and daemon are Blue Gene-specific,
- ZOID call forwarding code assumes that the architecture of the compute nodes and I/O forwarding nodes is the same, which does not have to be the case on machines targeted by the IOFSL project,

- the general-purpose function call forwarding infrastructure of ZOID could prevent some ZOIDFS-specific optimizations, such as pipelining (discussed below),
- using ZOID would add an external dependency to the IOFSL project, and we wanted to reduce the number of such dependencies to the minimum.

The call forwarding infrastructure being implemented uses Sun XDR for platform-independent marshalling and demarshalling of function arguments and results. On homogeneous systems, it will be possible to bypass the XDR encoding.

The I/O daemon currently in place is a prototype, featuring a simple design that can only handle one ZOIDFS call at a time. The production version will manage multiple ongoing operations simultaneously.

Complete function call requests are dispatched to the I/O forwarding nodes using BMI's unexpected messages—an asynchronous API for eager messages. BMI has a restriction on the maximum size of an unexpected message. The exact size is protocol-dependent, and is at least 8 KB for all current implementations; we expect this to be large enough for metadata operations, for requests for I/O, and even to carry the data for small write operations. Each unexpected request message from a compute node is followed by an expected response message from the I/O forwarding node, containing success/failure indicator, output data (if any), etc.

4.1 Pipelining

Given that one I/O forwarding node must be able to handle requests from multiple compute nodes, overloading an I/O node is always a concern. Other I/O forwarding infrastructures, such as IBM's CIOD or ZeptoOS ZOID, put a maximum size on data requests and require the client side to split up overly large requests into smaller operations. The I/O forwarding daemon then receives multiple requests that appear to be independent.

We have decided instead to expose the full request size to the I/O forwarding daemon and to let the daemon decide how the data transfers should be handled based on the current load, the amount of memory available, etc. This approach will give the daemon the contextual information needed to perform various optimizations.

In particular, large requests will be pipelined. Once the daemon has negotiated with the client the size of individual sub-operations, data transfer will proceed in chunks of the negotiated size. We expect to negotiate not only the size of individual sub-operations, but also the number of such sub-operations allowed to be posted by the client at any one time; this will take form of a credit system. Because the daemon knows of the complete request from the start, it will be able to overlap (pipeline) calls to the filesystem client with data transfers to/from the compute node. That would not be possible with other I/O forwarding infrastructures mentioned above, because for them the client is the one in charge of the progress of the whole request, and it cannot issue the next individual sub-operation before the previous one has fully completed (the API is blocking).

Numerous research avenues are made available through the pipelining facility. One such possibility would be the use of out-of-order data delivery techniques to enhance locality of access (e.g., if some of the data has been cached from previous operations). Another possibility would be for the daemon to notify the filesystem to expect more I/O operations in the near future; this could influence the filesystem caching/prefetching algorithms.

4.2 Fault Detection and Tolerance

With a possibility of multiple individual sub-operations being posted by a client at any time, gracefully recovering from file I/O errors can become problematic. What should happen to the remaining chunks if an earlier chunk failed? The easiest solution is to keep transferring the remaining chunks in order to satisfy the pending receives; otherwise, those receives would have to be canceled. Obviously, transferring can be potentially a lot slower than canceling, but we expect I/O errors to be rare, so the time it takes to recover from them is mostly irrelevant. Besides, as an optimization we could possibly send special, short null-messages to satisfy the receives.

We expect the reading part of the fault handling code to be exercised fairly well in day-to-day operations, because that code will also be used for handling short reads, i.e., those where less data is available from the file than was requested.

To improve the overall system resiliency, we plan to implement timeouts and retransmit messages for in-progress operations if they have not been acknowledged within a predefined time window. We will consider rerouting requests to alternative I/O forwarding nodes should the primary node stop responding.

A separate issue is the integrity of data during the transfers between the compute nodes and the I/O forwarding nodes. Contemporary networks generally provide some data integrity checks by default; however, there are indications in the HEC community that these checks are not sufficiently strong. In light of this, we have considered the following options:

- calculate complete CRCs of all messages,
- calculate CRCs of message headers only,
- do nothing.

No final decisions have been made, but we are leaning towards doing nothing by default in production environments for performance reasons, and calculating complete CRCs of all messages as an option. We expect that the performance impact of calculating complete CRCs would be too high to have it on by default, especially on architectures with relatively slow CPU cores and fast networks, such as the Blue Gene, but we will revisit this issue once the schemes have been implemented and their performance has been evaluated.

While we expect that calculating CRCs of message headers only would be a viable option performance-wise, the usefulness of such limited CRCs is questionable. Headers constitute a very small fraction of bytes transferred, so, assuming a uniform distribution of corruptions, message bodies are a lot more likely to get corrupted, and those corruptions would not be noticed under this scheme. On the other hand, a silent corruption of a message header could seriously confuse the protocol stack, so protecting them more strongly than the message bodies could be worthwhile.

We are currently planning to implement calculating the CRCs inside the BMI layer. There are several options regarding what to do with messages that fail their CRC check on the receiver side. The simplest choice would be to simply drop them, in an expectation that this would result in a timeout and a retransmit in a higher layer of the network stack. BMI could also return an error condition on the receiver side, but it is unclear how useful that would be. If the message is corrupted, its headers cannot be trusted, so there might be no way to find out what node the message in question arrived from (unless, as indicated earlier, the header was protected with a separate CRC).

5 Interface on the Client

5.1 ROMIO

We follow our PVFS approach here. A ROMIO driver will be able to turn MPI file views and datatypes into offset-list pairs, thereby delivering good noncontiguous I/O performance. We can utilize other ROMIO optimizations as well: two-phase collective I/O and data sieving (for reads only, as writes would require locking).

The initial ZOIDFS driver for ROMIO went into MPICH2 trunk in subversion revision 2944. This version does not yet have list I/O, but does let us test out contiguous I/O benchmarks. MPI-IO metadata operations (open, create, resize) are supported. We have done most of our code testing using the Blue Gene-specific infrastructure, with some recent tests using IOFSL.

ZOIDFS for ROMIO is not available in a released version of MPICH2, but MPICH2-1.1 will contain this work.

5.2 SYSIO

libsysio [7] is expected to be the primary means of redirecting application's POSIX file I/O calls to the ZOIDFS API on platforms with a microkernel lacking a proper VFS layer, such as Catamount on Cray XT, and possibly also IBM CNK on Blue Gene.

The work on implementing a ZOIDFS backend for libsysio has not started yet.

5.3 FUSE

For systems supported by the FUSE [2] project (Linux, OpenSolaris and FreeBSD, among others), another way of redirecting POSIX calls is available.

The FUSE kernel module enables us to transparently intercept POSIX file operations without requiring any application modifications. These operations are subsequently directed to the ZOIDFS library. Special care has to be taken to match the stateful API of FUSE with the stateless ZOIDFS API. This work is mostly complete.

Although FUSE currently provides the most transparent and user-friendly option, it might not be the most efficient choice. Unlike the alternative methods described above—which execute all I/O handling in userspace—I/O operations using FUSE first travel to the kernel where they are redirected back to a userspace library. Further testing is needed to determine if the overhead incurred by this userspace to kernel roundtrip is acceptable.

6 Interfacing to Filesystems on the Server

There are a number of possible options for interfacing to underlying file systems on the server (I/O forwarding node) side, including making direct POSIX calls and relying on the OS to vector file operations to appropriate file systems, making direct PVFS library calls, implementing the server in terms of the ZOIDFS API and then providing local data access versions of this API, and implementing the server on top of libsysio and using its facilities.

We currently believe the use of libsysio as our API to access file system resources on the I/O forwarding node is our best long-term option. This approach has the advantages of using existing production code, maintaining the majority of functionality in user space where it is most easily debugged, and being a nice match between the APIs—the libsysio and ZOIDFS APIs are very similar.

However, additional development work is needed on libsysio before it will be ready for use in this mode, particularly in the implementation of handle-based access. For file systems or interfaces that do not export handles directly (e.g., POSIX), this implementation is nontrivial. While this development is underway, we are pursuing an alternative mechanism for file system access on the server side.

The ZOIDFS API provides a stateless interface to files much like PVFS, so the PVFS driver within ZOIDFS is a direct mapping from ZOIDFS calls to PVFS calls. ZOIDFS uses stateless handles (an opaque 128-bit field) to refer to objects within the filesystem; these must be mapped to the appropriate filesystem-specific structures. PVFS uses file system identifiers (a 32-bit field) and object identifiers (a 64-bit field) to identify objects within the filesystem. These can be encapsulated in the ZOIDFS handle, but must be converted back to PVFS fs-ids and objects with each call. At present, ZOIDFS calls are blocking, so the PVFS driver uses the blocking PVFS calls.

For testing purposes, a direct reimplementation of the ZOIDFS API using POSIX calls is being developed. It can optionally rely on an external database server to emulate persistent and stateless file handles for files located on a normal POSIX filesystem. If file handles will not be shared between nodes, a local database can be used instead, simplifying deployment.

Implementing a stateless API on top of a stateful API (such as the POSIX I/O functions) requires an extra layer of resource management; in particular, due to the fact that files cannot be kept open indefinitely, a form of garbage collection is employed to close inactive files. However, as handles in the ZOIDFS API remain valid as long as the file exists, the implementation needs to be able to reopen the file when presented with the original ZOIDFS handle. Unfortunately, as POSIX only allows opening a file by specifying the full path name, a mechanism is needed to reconstruct the original path name given the ZOIDFS handle. Our implementation does this by recording the mapping between a ZOIDFS handle and the pathname in the external database.

The use of an additional database is but one approach to solving this problem. libsysio pursues another path by attempting to compress enough information into the handle to reconstruct the path name.

7 Online Resources

The following resources dedicated to the IOFSL project are available online:

- Project Website [3],
- Code Repository [4],
- Document Repository [5].

References

[1] P. Carns, W. Ligon III, R. Ross, and P. Wyckoff. BMI: A network abstraction layer for parallel I/O. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, Workshop on Communication Architecture for Clusters*, Denver, CO, April 2005.

[2] FUSE: Filesystem in userspace. http://fuse.sourceforge.net/.

[3] IOFSL: I/O forwarding scalable layer. http://www.iofsl.org/.

[4] The IOFSL code repository. https://svn.mcs.anl.gov/repos/ZeptoOS/branches/zoidfs-bmi/.

[5] The IOFSL document repository. https://svn.mcs.anl.gov/repos/iofsl/papers/.

[6] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman. ZOID: I/O-forwarding infrastructure for petascale architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 153–162, Salt Lake City, UT, Feb. 2008.

[7] The SYSIO library. http://libsysio.sourceforge.net/.

[8] R. Ross, J. Nunez, P. Beckman, J. Bent, G. Grider, S. Poole, L. Ward, and P. Wyckoff. FASTOS (LAB 07-23) Proposal for a common scalable HEC I/O forwarding layer.

[9] The ZeptoOS project. http://www.zeptoos.org/.